# Fast Maintenance of 2-hop Labels for Shortest Distance Queries on Fully Dynamic Graphs

Anonymous

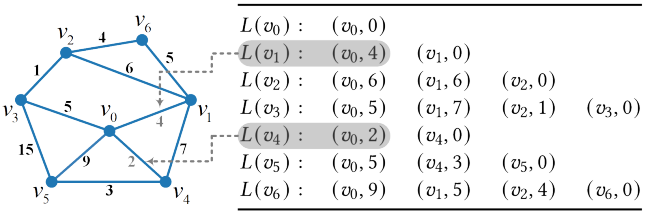



**Figure 1: A running example of 2-hop labels.**

## ABSTRACT

**ACM Reference Format:**
Anonymous. 2025. Fast Maintenance of 2-hop Labels for Shortest Distance Queries on Fully Dynamic Graphs. In *Proceedings of the 2025 International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Querying shortest distances is an essential step of various graph mining tasks, *e.g.,* the algorithms for information retrieval in knowledge graphs [11, 20, 22, 24] and crowd-sourcing in social networks [10, 16, 23] require frequently querying shortest distances in the computing process. Due to this practical usefulness, a lot of work [1, 3, 6, 7, 12, 15, 17, 18, 26] has been done to explore the indexing approach to fast querying shortest distances on graphs. In particular, 2-hop labels [2, 3, 7, 12, 13, 15] are state-of-the-art shortest distance indexes that have been intensively studied in recent years.

We introduce 2-hop labels [7] as follows. For each vertex $s$, there is a set $L(s)$ of labels. Each label in $L(s)$ is a two-element tuple that contains (i) a vertex $h$ that is called a hub of $s$; and (ii) the shortest distance between $h$ and $s$, *i.e.,* $d(h, s)$. We generate labels that meet the *2-hop cover constraint* [7]: for each pair of vertices $s$ and $t$, there is at least one vertex $h$ that is (i) a hub of both $s$ and $t$; and (ii) in a shortest path between $s$ and $t$. Then, we can query the shortest distance between $s$ and $t$ as $d(h, s) + d(h, t)$, *e.g.,* in Figure 1, we can query the shortest distance between $v_1$ and $v_4$ as 6 using two labels $(v_0, 4) \in L(v_1)$ and $(v_0, 2) \in L(v_4)$. The recent work [3, 12, 13, 15] shows that we can use 2-hop labels to query shortest distances between a pair of vertices on a large graph with millions of vertices and edges within milliseconds.

On the other hand, real graphs often evolve over time, in terms of both topology and weight [5, 19, 21, 29]. For example, the proximity between two users in a social network may change over time due to their activities (likes, followers, tags, *etc.*) [21, 29]. Even though the existing work [3, 12, 13, 15] can generate 2-hop labels for a large graph with millions of vertices and edges, it is inefficient to re-generate labels whenever a graph change occurs. As a result, some work [4, 8, 27–29] has recently been conducted to maintain 2-hop labels in dynamic scenarios, for ensuring the correctness of distance query results on the changed graph.

Specifically, the existing work [4, 8, 27–29] generally considers the following types of graph changes: vertex/edge insertions and deletions, and edge weight increases and decreases. As discussed in [27–29], vertex/edge insertions and deletions can be considered as special cases of edge weight changes. In particular, vertex/edge deletions are equivalent to increasing edge weights to infinity, while vertex/edge insertions can be regarded as edge weight decreases in a similar way [27–29]. Therefore, like the above existing studies, we mainly focus on two types of graph changes: edge weight increases and decreases, without losing the generality of our work.

**Motivation:** To our knowledge, Akiba *et al.* [4] first explore the topic of 2-hop label maintenance on dynamic graphs. They propose an algorithm, which we refer to as DePLL, that can maintain labels after an edge weight decrease. Later, D'angelo *et al.* [8] complete this work by proposing another algorithm, which we refer to as InPLL, that can maintain labels after an edge weight increase. They show the combination of DePLL and InPLL as the first solution to maintaining 2-hop labels in the fully dynamic scenario where edge weights may alternately increase and decrease.

Nevertheless, neither DePLL nor InPLL is efficient. First, due to the lack of the rank pruning technique in [12], DePLL conducts a lot of unnecessary attempts to generate new labels. Second, due to the lack of efficient methods to identify labels affected by edge weight increases, InPLL performs a large number of breadth first searches over the whole graph to update such affected labels.

To address the inefficiency issue of DePLL and InPLL, Zhang *et al.* [27] recently develop two different algorithms, DeAsyn and InAsyn, for efficiently maintaining labels in edge weight decrease and increase cases, respectively. By incorporating the rank pruning technique [12] into the maintaining process, as well as by recording information that helps us to efficiently identify labels affected by edge weight increases, DeAsyn and InAsyn achieve the state-of-the-art performance for maintaining labels in edge weight decrease and increase cases, respectively. In particular, a recent experimental study [29] shows that DeAsyn and InAsyn can be up to an order of magnitude faster than DePLL and InPLL, respectively.

However, we note that DeAsyn and InAsyn cannot be combined to deal with the fully dynamic case where edge weights may alternately increase and decrease, and can only be used when edge weights solely decrease or increase, respectively. The reason is that DeAsyn cannot update all outdated labels, and could leave some outdated labels with incorrectly large distance values out of control after an edge weight decrease maintenance, while these incorrectly


*SIGMOD '25, May 15, 2025, Berlin, Germany*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

large values may become incorrectly small after a later edge weight increase, and then induce incorrect distance query results.

Moreover, even after repairing DeAsyn, it is still not fast enough to apply DeAsyn and InAsyn to the fully dynamic case. Particularly, we note that these two algorithms conduct an unnecessarily large number of distance queries in the maintaining process. As a result, there is still no fast solution to maintaining 2-hop labels in the fully dynamic scenario to date, despite its distinct practical usefulness.

**Contributions:** We address the above issues by proposing several novel and efficient algorithms. The contributions are as follows.

- First, we show that the state-the-art DeAsyn and InAsyn cannot be combined to deal with the fully dynamic scenario, due to the aforementioned drawback of DeAsyn. To address this issue, we repair it as RepairedDeAsyn, by updating all outdated labels. Furthermore, we point out that, DeAsyn, RepairedDeAsyn and InAsyn unnecessarily conduct $O(\Upsilon \cdot d_a^2)$ distance queries, and consequently still suffer from the inefficiency issue, where $\Upsilon$ is the number of updated labels, and $d_a$ is the average degree of vertices associated with these labels.
- The state-of-the-art algorithms cannot be trivially modified to address the above inefficiency issue. To overcome this challenge, we propose two new algorithms, FastDeM and FastInM, for maintaining 2-hop labels after edge weight decreases and increases, respectively. Both algorithms update all outdated labels, and together can deal with fully dynamic scenarios. Moreover, both algorithms remarkably differ from state-the-art algorithms by employing a novel and non-trivial hub-sorted label update method with distance query result memorization, without increasing the space complexity. Specifically, first, both algorithms decouple the mixed processes of updating labels with different hubs in state-of-the-art algorithms, resulting in a hub-sorted label update method. Second, different from existing 2-hop label maintenance [4, 8, 27] and generation [3, 9, 12] algorithms that all conduct a distance query before each label update, both algorithms dissociate distance queries from label updates, and use hash tables to record and update some distance query results, without actually conducting these queries, in the maintaining process. By applying these novelties, both algorithms reduce the above complexity of $O(\Upsilon \cdot d_a^2)$ to $O(\Upsilon \cdot d_a)$.

# 2 PRELIMINARIES

## 2.1 Problem Formulation

Given an edge-weighted graph $G(V, E, w)$, where $V$ is the set of vertices, $E$ is the set of edges, and $w$ is a function which maps each edge $(u, v) \in E$ between a pair of vertices $u$ and $v$ to a positive value $w(u, v)$ that we refer to as edge weight. We use $N(v)$ to denote the set of adjacent vertices of $v$, and use $d(v_i, v_j)$ to denote the shortest distance between a pair of vertices $v_i$ and $v_j$.

As discussed in the beginning of this paper, querying shortest distances is useful in various scenarios. 2-hop labels [7] enable us to fast query shortest distances, and have been intensively studied in recent years [2, 3, 12, 13, 15]. In the 2-hop labeling framework, for each vertex $v \in V$, there is a set $L(v)$ of labels. Each label in $L(v)$ is a two-element tuple $(u, d(u, v))$, where $u$ is a vertex and $d(u, v)$ is the shortest distance between $u$ and $v$. If $(u, d(u, v)) \in L(v)$, then we denote $L(v)[u] = d(u, v)$, and consider $u$ as a hub of $v$. We use $C(v)$ to denote the set of hubs of $v$, *i.e.,* $C(v) = \{u | (u, d(u, v)) \in L(v)\}$. We further use $L$ to denote the total set of 2-hop labels. We let $L$ satisfy the following 2-hop cover constraint [7].

Definition 1 (2-hop Cover Constraint [7]). *A set $L$ of 2-hop labels satisfies the 2-hop cover constraint if, for every pair of vertices $v_i$ and $v_j$, there is a common hub vertex $u \in C(v_i) \cap C(v_j)$ in a shortest path between $v_i$ and $v_j$, i.e., $d(v_i, v_j) = d(u, v_i) + d(u, v_j)$.*

With a set of 2-hop labels that satisfies the above constraint, we can query $d(v_i, v_j)$ using $L$ based on the following equation.

$$d(v_i, v_j) = \min_{u \in C(v_i) \cap C(v_j)} d(u, v_i) + d(u, v_j). \quad (1)$$

For instance, in Figure 1, we can query $d(v_1, v_4) = 4 + 2 = 6$ using two labels $(v_0, 4) \in L(v_1)$ and $(v_0, 2) \in L(v_4)$.

On the other hand, real graphs often evolve as time passes. Like the related work [27–29], we mainly focus on edge weight increases and decreases, since edge weight changes generalize vertex/edge insertions and deletions. Also like the existing work [4, 8, 27–29], we study the following 2-hop label maintenance problem.

Problem 1. *Given an edge-weighted graph $G(V, E, w)$ and a set $L$ of 2-hop labels that satisfies the 2-hop cover constraint, after an edge weight increase or decrease on $G$, the problem is to maintain $L$ such that we can use the maintained $L$ to correctly query the shortest distance between every pair of vertices on the changed $G$.*

In particular, like [8], we will develop algorithms to consecutively update $L$ for each change in the fully dynamic scenario where edge weights may alternately decrease and increase.

## 2.2 Generation of 2-hop Labels and PPR

Like the existing work on 2-hop label maintenance [4, 8, 27–29], we employ the Pruned Landmark Labeling (PLL) algorithm [3] to generate $L$ that satisfies the 2-hop cover constraint, since PLL is the core of many state-of-the-art 2-hop labeling techniques [13, 15, 25], and the other methods for generating 2-hop labels, such as PSL [12] and BVC-PLL [9], do not suit weighted graphs.

PLL assumes that vertices are ranked in such a way that higher-ranked vertices are likely to be in more shortest paths, and then uses higher-ranked vertices as hubs of lower-ranked ones, *i.e.,* a vertex can only have higher-ranked hubs than itself, hopefully resulting in a small label size, since fewer labels consume less memory to store and also enable faster queries by Equation (1). The state-of-the-art work [3, 12–15] shows that ranking vertices by their degrees from large to small induces a high indexing performance, since vertices with larger degrees are likely to be in more shortest paths. We follow this ranking strategy. Let $V = \{v_0, \cdots, v_{|V|-1}\}$. We use $r(v_i)$ to denote the rank of $v_i$. Let $deg(v_0) \geq \cdots \geq deg(v_{|V|-1})$, and $r(v_0) > \cdots > r(v_{|V|-1})$, where $deg(v_i)$ is the degree of $v_i$.

The idea of PLL is to dynamically generate labels in a decreasing order of hub ranks via Dijkstra-style searches, and to use generated labels to query distances for possibly pruning these searches. Different from the original PLL, we further use the Pruning Point Record (PPR) [27] data structure to record the pruning information in PLL. Such information helps us to quickly identify labels affected by edge weight increases. For a pair of vertices $v_i$ and $v_j$, $PPR[v_i, v_j]$ is a

**Algorithm 1** The PLL algorithm incorporated with PPR

**Input:** a graph $G(V, E, w)$ **Output:** $L$ and $PPR$

1: Initialize $L = PPR = \emptyset$
2: **for** each sorted vertex $u \in V$ **do**
3: Initialize $Q = \emptyset$, $d(u) = 0$, $d(v) = NIL$ for each $v \in V \setminus u$
4: Insert $u$ into $Q$ with the priority value of $d(u)$
5: **while** $Q \neq \emptyset$ **do**
6: Pop $v$ out of $Q$ with the priority value of $d(v)$
7: **if** $r(u) \geq r(v)$ **then**
8: **if** $Query(u, v, L) \leq d(v)$ **then**
9: $PPR[v, h_c].push(u)$, $PPR[u, h_c].push(v)$; Continue to Line 5
10: Insert $(u, d(v))$ into $L(v)$
11: **for** each vertex $x \in N(v)$ **do**
12: **if** $d(x) == NIL$ **then**
13: Insert $x$ into $Q$ with the priority value of $d(x) = d(v) + w(x, v)$
14: **else if** $d(x) > d(v) + w(x, v)$ **then**
15: Update $x$ in $Q$ with the priority value of $d(x) = d(v) + w(x, v)$
16: Return $L$ and $PPR$

**Table 1: The generated $PPR$ for the example in Figure 1.**

| $PPR[v_1, v_0]$ | $PPR[v_2, v_0]$ | $PPR[v_3, v_0]$ | $PPR[v_4, v_0]$ | $PPR[v_5, v_0]$ |
|---|---|---|---|---|
| $\{v_4, v_5\}$ | $\{v_5\}$ | $\{v_5\}$ | $\{v_1\}$ | $\{v_1, v_2, v_3\}$ |

set of vertices. We will explain the meaning of PPR after describing PLL incorporated with PPR (Algorithm 1) as follows.

**The PLL algorithm incorporated with PPR:** Given a graph $G$, PLL initializes $L = PPR = \emptyset$ (Line 1). Then, it sequentially processes each $u \in V$ in a decreasing order of vertex ranks, *i.e.,* from $v_0$ to $v_{|V|-1}$ (Line 2). It initializes a min priority queue $Q$, and sets $d(u) = 0$ and $d(v) = NIL$ for each $v \in V \setminus u$ (Line 3). Subsequently, it inserts $u$ into $Q$ with the priority of $d(u)$ (Line 4). While $Q \neq \emptyset$ (Line 5), it pops $v$ out of $Q$ with the priority of $d(v)$ (Line 6). If the rank of $u$ is no lower than the rank of $v$ (Line 7), it conducts the following processes. If the queried distance between $u$ and $v$ using $L$ by Equation (1) is no larger than $d(v)$, it inserts $u$ into $PPR[v, h_c]$, and also inserts $v$ into $PPR[u, h_c]$, where $h_c \in C(u) \cap C(v)$ is the common hub responsible for the queried distance between $u$ and $v$, and then continues the while loop (Line 9). If the queried distance is larger than $d(v)$, then it inserts a label $(u, d(v))$ into $L(v)$ (Line 10). It processes each vertex $x \in N(v)$ as follows (Line 11). If $d(x) == NIL$ (Line 12), then it inserts $x$ into $Q$ with the priority of $d(x) = d(v) + w(x, v)$ (Line 13). If $d(x) \neq NIL$ and $d(x) > d(v) + w(x, v)$ (Line 14), then it updates $x$ in $Q$ with the priority of $d(x) = d(v) + w(x, v)$ (Line 15). In the end, it returns the generated $L$ and $PPR$ (Line 16).

**An example of PLL:** In Figure 1, first, consider the loop of $u = v_0$ in Line 2, it inserts $(v_0, 0)$, $(v_0, 4)$, $(v_0, 6)$, $(v_0, 5)$, $(v_0, 2)$, $(v_0, 5)$, $(v_0, 9)$ into $L(v_0), \cdots, L(v_6)$, respectively, and inserts no element into $PPR$, since the queried distance between $v_0$ and $v_i$ for each $i \in [0, 6]$ in Line 8 is infinity. Then, in the loop of $u = v_1$ in Line 2, it inserts $(v_1, 0)$, $(v_1, 6)$, $(v_1, 7)$, $(v_1, 5)$ into $L(v_1), L(v_2), L(v_3), L(v_6)$, respectively. When it tries to insert $(v_1, 7)$ into $L(v_4)$, it queries the distance between $v_1$ and $v_4$ as 6 using two inserted label $(v_0, 4) \in L(v_1)$ and $(v_0, 2) \in L(v_4)$, and hence does not insert $(v_1, 7)$ into $L(v_4)$, but inserts $v_1$ into $PPR[v_4, v_0]$, and also inserts $v_4$ into $PPR[v_1, v_0]$. Similarly, it does not insert $(v_1, 22)$ into $L(v_5)$, but inserts $v_1$ into $PPR[v_5, v_0]$, and also inserts $v_5$ into $PPR[v_1, v_0]$. Subsequently, in the loops of $u = v_2, \cdots, v_6$ in Line 2, it inserts more elements into $L$ and $PPR$ as shown in Figure 1 and Table 1, respectively.

**The meaning of PPR:** Algorithm 1 constructs $PPR$ in Line 9, where it first inserts $u$ into $PPR[v, h_c]$, which means that the common hub $h_c \in C(u) \cap C(v)$ is responsible for pruning the spread of hub $u$ from a neighbor of $v$ to $v$, and then symmetrically, also inserts $v$ into $PPR[u, h_c]$. Thus, the fact that $v_k \in PPR[v_i, v_j]$ indicates that either $v_j \in C(v_i) \cap C(v_k)$ is responsible for pruning the spread of hub $v_k$ from a neighbor of $v_i$ to $v_i$ (which means that $r(v_k) > r(v_i)$), or $v_j$ is responsible for pruning the spread of hub $v_i$ from a neighbor of $v_k$ to $v_k$ (which means that $r(v_i) > r(v_k)$). The pruning information recorded by PPR helps us to quickly identify labels affected by an edge weight increase, and plays a critical role in achieving a high efficiency of maintaining 2-hop labels.

## 3 DEFECTS OF EXISTING ALGORITHMS

A recent experimental study [29] shows that DeAsyn and InAsyn [27] have the state-of-the-art performance for maintaining 2-hop labels in edge weight decrease and increase scenarios, respectively. In this section, we show that these two algorithms cannot be combined to deal with the fully dynamic scenario, and also repair DeAsyn to address this issue. Moreover, we analyze that these two algorithms conduct an unnecessarily large number of distance queries, and consequently still suffer from the inefficiency issue.

### 3.1 The inability of DeAsyn to deal with the fully dynamic scenario

We observe that DeAsyn may leave outdated labels out of control after the maintenance, and as a result cannot deal with the fully dynamic scenario. Particularly, suppose that the weight of edge $(a, b)$ decreases from $w_0$ to $w_1$, *i.e.,* $w_0 > w_1$. If there is an outdated label-contained distance value $L(v)[u]$ that can be decreased by spreading hub $u$ to $v$ via the new edge $(a, b)$, then DeAsyn conducts this spread and updates $L(v)[u]$ only when the queried distance between $u$ and $v$ by Equation (1) is larger than the to-be-updated distance value, but does not do so otherwise. In this sub-section, we first modify DeAsyn to update every such outdated value, no matter whether the queried distance is larger than the to-be-updated distance value or not, resulting in RepairedDeAsyn (Algorithm 2). Then, we provide an example to illustrate that DeAsyn cannot, but RepairedDeAsyn can, deal with the fully dynamic scenario.

**The RepairedDeAsyn algorithm:** The idea of RepairedDeAsyn is to update all outdated label-contained distance values by spreading hubs via the new $(a, b)$, first for labels of $a$ and $b$, and then neighbors of $a$ and $b$, and then neighbors of neighbors, *etc.*

The algorithm inputs the updated graph $G(V, E, w)$, $L$, $PPR$ and $(a, b)$. It initializes two empty sets: $CL^c$ and $CL^n$ (Line 1). It uses $CL^c$ to record new labels of $b$ as follows. For each $(v, d_{va}) \in L(a)$ (Line 2), if $r(v) \geq r(b)$ (Line 3), then it checks whether the queried distance between $v$ and $b$ is larger than $d_{va} + w_1$. If it is, then it sets $L(b)[v] = d_{va} + w_1$, and pushes $(b, v, d_{va} + w_1)$ into $CL^c$ (Line 5). Otherwise, it conducts the above step when $v \in C(b)$ & $L(b)[v] > d_{va} + w_1$ (Line 7), and then inserts $v$ into $PPR[b, h_c]$, and also inserts $b$ into $PPR[v, h_c]$, where $h_c$ is the common hub responsible for the queried distance between $v$ and $b$. The condition that $v \in C(b)$ & $L(b)[v] > d_{va} + w_1$ means that $v$ is already a hub of $b$, and $L(b)[v]$ is larger than, but should be decreased to, $d_{va} + w_1$. It uses $CL^c$ to record new labels of $a$ similarly (Lines 10-17).

Subsequently, while $CL^c \neq \emptyset$, it iteratively uses the *ProDecrease* procedure to update more labels (Line 18). $CL^c$ and $CL^n$ are the sets of updated labels in the last and current iterations, respectively.

**Algorithm 2** The RepairedDeAsyn algorithm

**Input:** the updated graph $G(V, E, w)$, $L$, $PPR$, $(a, b)$ // $w_0 > w_1$
**Output:** the maintained $L$ and $PPR$

1: $CL^c = CL^n = \emptyset$
2: **for** each label $(v, d_{va}) \in L(a)$ **do**
3: **if** $r(v) \geq r(b)$ **then**
4: **if** $Query(v, b, L) > d_{va} + w_1$ **then** // $w(a, b) = w_1$
5: $L(b)[v] = d_{va} + w_1, CL^c.push((b, v, d_{va} + w_1))$
6: **else**
7: **if** $v \in C(b)$ & $L(b)[v] > d_{va} + w_1$ **then**
8: $L(b)[v] = d_{va} + w_1, CL^c.push((b, v, d_{va} + w_1))$
9: $PPR[b, h_c].push(v), PPR[v, h_c].push(b)$
10: **for** each label $(v, d_{vb}) \in L(b)$ **do**
11: **if** $r(v) \geq r(a)$ **then**
12: **if** $Query(v, a, L) > d_{vb} + w_1$ **then**
13: $L(a)[v] = d_{vb} + w_1, CL^c.push((a, v, d_{vb} + w_1))$
14: **else**
15: **if** $v \in C(a)$ & $L(a)[v] > d_{vb} + w_1$ **then**
16: $L(a)[v] = d_{vb} + w_1, CL^c.push((a, v, d_{vb} + w_1))$
17: $PPR[a, h_c].push(v), PPR[v, h_c].push(a)$
18: **while** $CL^c \neq \emptyset$ **do** $ProDecrease(CL^c, CL^n), CL^c = CL^n, CL^n = \emptyset$
19: Return $L$ and $PPR$

**Procedure** $ProDecrease(CL^c, CL^n)$
20: **for** each $(u, v, d_u) \in CL^c$ **do**
21: **for** each $u_n \in N(u)$ **do**
22: **if** $r(v) > r(u_n)$ **then**
23: **if** $Query(v, u_n, L) > d_{new} = d_u + w(u, u_n)$ **then**
24: $L(u_n)[v] = d_{new}, CL^n.push((u_n, v, d_{new}))$
25: **else**
26: **if** $v \in C(u_n)$ & $L(u_n)[v] > d_{new}$ **then**
27: $L(u_n)[v] = d_{new}, CL^n.push((u_n, v, d_{new}))$
28: $PPR[u_n, h_c].push(v), PPR[v, h_c].push(u_n)$

To perform these iterations, it sets $CL^c = CL^n$ and $CL^n = \emptyset$ after using $ProDecrease$ in each iteration. In $ProDecrease$, for each $(u, v, d_u) \in CL^c$ (Line 20), it checks each neighbor $u_n$ of $u$ (Line 21). If $r(v) > r(u_n)$, it checks whether the queried distance between $v$ and $u_n$ is larger than $d_{new} = d_u + w(u, u_n)$ (Line 23). If it is, then it updates $L(u_n)[v] = d_{new}$, and pushes $(u_n, v, d_{new})$ into $CL^n$ (Line 24). Otherwise, it conducts the above step when $v \in C(u_n)$ & $L(u_n)[v] > d_{new}$, and then inserts $v$ into $PPR[u_n, h_c]$, and also inserts $u_n$ into $PPR[v, h_c]$ (Line 28), where $h_c$ is the common hub responsible for the queried distance between $v$ and $u_n$. In the end, it returns the updated $L$ and $PPR$ (Line 19).

DeAsyn **cannot, but** RepairedDeAsyn **can, deal with the fully dynamic scenario:** RepairedDeAsyn is different from DeAsyn in that RepairedDeAsyn conducts the additional checks in Lines 7, 15 and 26 to update outdated label-contained distance values. That is to say, by removing Lines 7-8, 15-16 and 26-27 in RepairedDeAsyn, we obtain DeAsyn, *i.e.,* Algorithm 3 in [27].

We use the following Figure 2 to illustrate that, due to the above difference, DeAsyn cannot, while RepairedDeAsyn can, deal with the fully dynamic scenario. There are five edge weight decreases, followed by two edge weight increases. All these seven edge weight changes play a critical role in this illustration.

We show that RepairedDeAsyn can be combined with InAsyn to deal with these changes as follows. First, $w(v_1, v_4)$ decreases from 7 to 5.6. RepairedDeAsyn re-spreads hub $v_1$ from itself to $v_4$ and $v_5$ to update two new labels $(v_1, 5.6) \in L(v_4)$, $(v_1, 8.6) \in L(v_5)$. Second, $w(v_4, v_5)$ decreases from 3 to 2. RepairedDeAsyn updates three new labels $(v_0, 4) \in L(v_5)$, $(v_1, 7.6) \in L(v_5)$, $(v_4, 2) \in L(v_5)$ by re-spreading hubs $v_0$, $v_1$ and $v_4$ from $v_4$ to $v_5$. Third, $w(v_0, v_1)$ decreases from 4 to 3. RepairedDeAsyn updates two labels $(v_0, 3) \in L(v_1)$ and $(v_0, 8) \in L(v_6)$. Fourth, $w(v_0, v_5)$ decreases from 9 to 3. RepairedDeAsyn updates a single label $(v_0, 3) \in L(v_5)$. Fifth, $w(v_4, v_5)$ decreases from 2 to 1. RepairedDeAsyn updates two labels $(v_1, 6.6) \in L(v_5)$ and $(v_4, 1) \in L(v_5)$. Notably, $d(v_1, v_5)$ is 6, not 6.6, at this moment. Nevertheless, we can tolerate $(v_1, 6.6) \in L(v_5)$, since it does not induce incorrect query results, and is a direct spread of another label $(v_1, 5.6) \in L(v_4)$ via the updated $(v_4, v_5)$, and as a result can be updated by a later edge weight increase maintenance. Sixth, $w(v_0, v_5)$ increases from 3 to 9. InAsyn or the later proposed FastInM updates no label (we refer the details of InAsyn to [27]). Seventh, $w(v_4, v_5)$ increases from 1 to 3. InAsyn or FastInM updates $(v_0, 5) \in L(v_5)$ and $(v_4, 3) \in L(v_5)$, and deactivates $(v_1, 6.6) \in L(v_5)$ to be $(v_1, \infty) \in L(v_5)$. All distances can be correctly queried after the maintenance. Hence, RepairedDeAsyn can be combined with InAsyn or FastInM to deal with the above fully dynamic case.

However, DeAsyn cannot do this. In particular, DeAsyn cannot update $(v_1, 7.6) \in L(v_5)$ to $(v_1, 6.6) \in L(v_5)$ after the 5th change, as the queried distance between $v_1$ and $v_5$: 6 is smaller than 6.6, due to the existence of $(v_0, 3) \in L(v_1)$ and $(v_0, 3) \in L(v_5)$. As a result, $(v_1, 7.6) \in L(v_5)$ remains. Different from the failed-to-update label $(v_1, 6.6) \in L(v_5)$, the remaining $(v_1, 7.6) \in L(v_5)$ is not a direct spread of another label $(v_1, 5.6) \in L(v_4)$ via the updated $(v_4, v_5)$, as $w(v_4, v_5) = 1$ after the 5th change, and $5.6 + 1 \neq 7.6$. To maintain labels for the 7th change, both InAsyn and the later proposed FastInM identify to-be-updated labels in $L(v_5)$ by checking whether these labels are direct spread of labels in $L(v_4)$ along $(v_4, v_5)$ (the corresponding step in InAsyn is in Line 6 of Algorithm 4 in [27]; a similar step is the if condition $(v, d_{va} + w_0) \in L(b)$ in Line 3 of the later proposed FastInM). Consequently, after the 7th change, neither InAsyn nor FastInM could deactivate the outdated label $(v_1, 7.6) \in L(v_5)$. The outdated distance value 7.6 is incorrectly large before the 7th change, but becomes incorrectly small after the 7th change. Due to this error, the queried distance between $v_1$ and $v_5$ is incorrectly 7.6 after the 7th change (the correct distance is 8). Thus, DeAsyn cannot be combined with InAsyn or the later proposed FastInM to deal with the above fully dynamic case.

## 3.2 The inefficiency of DeAsyn and InAsyn

We further observe that both DeAsyn and InAsyn conduct an unnecessarily large number of distance queries, and consequently still suffer from the inefficiency issue. Specifically, let $\Upsilon$ be the number of updated labels, and let $d_a$ be the average degree of vertices associated with these labels. Both DeAsyn and InAsyn unnecessarily conduct $O(\Upsilon \cdot d_a^2)$ distance queries in the maintaining process.

We take the following example of edge weight decrease maintenance to show that DeAsyn (or RepairedDeAsyn) conducts $O(\Upsilon \cdot d_a^2)$ distance queries. Since InAsyn updates labels in a similar iterative way with DeAsyn, *i.e.,* the iterative calls of procedures in InAsyn [27] to update labels is similar to the iterative call of $ProDecrease$ in DeAsyn, we omit a similar illustration for InAsyn.

Consider the graph in Figure 3, where $M \gg \Upsilon \gg d_a^2$; $w(v_0, v_1) = 2M$; $w(v_1, v_i) = 1$ for each $i \in [2, d_a + 1]$; there is a simple path between $v_i$ and $v_j$ for every pair of $i \in [2, d_a + 1]$ and $j \in [d_a + 2, \Upsilon]$, and this path contains $i - 1$ edges and has a total weight of $d_a - i + 2$. Suppose that $w(v_0, v_1)$ decreases from $2M$ to $M$. DeAsyn and RepairedDeAsyn maintain labels as follows. Initially, they update $L(v_1)[v_0] = M$ and $L(v_i)[v_0] = M + 1$ for each $i \in [2, d_a + 1]$. Subsequently, for a certain $j \in [d_a + 2, \Upsilon]$, they sequentially update

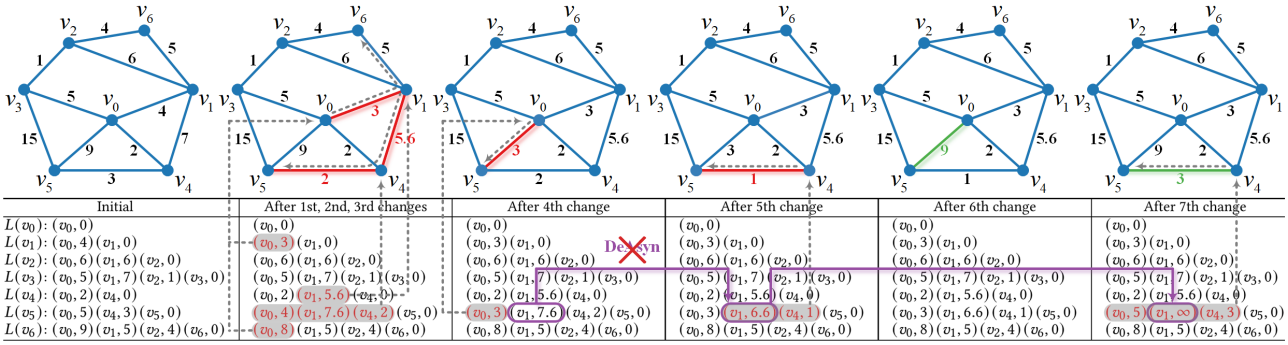



**Figure 2: An example of the fully dynamic scenario (red: weight decreases; green: weight increases).**

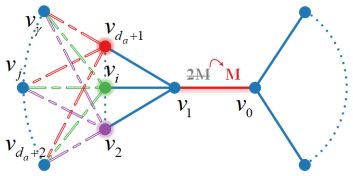



**Figure 3: An example of $O(\Upsilon \cdot d_a^2)$ distance queries.**

$L(v_j)[v_0] = M + 1 + d_a - i + 2$ through the path between $v_i$ and $v_j$ for each $i \in [2, d_a + 1]$. As a result, they update label-contained distance values $O(\Upsilon \cdot d_a)$ times. Notice that, each label update in an iterative call of *ProDecrease* induces $O(d_a)$ distance queries in the next iterative call of *ProDecrease*. Consequently, DeAsyn and RepairedDeAsyn conduct $O(\Upsilon \cdot d_a^2)$ distance queries.

On the other hand, each distance query by Equation (1) takes $O(\delta)$ time [14, 15], where $\delta$ is the average number of labels associated with each vertex. Thus, it costs $O(\Upsilon \cdot d_a^2 \cdot \delta)$ to conduct the above distance queries. The cost of conducting distance queries is a major part of the costs of DeAsyn and InAsyn. For example, DeAsyn (or RepairedDeAsyn) has a time complexity of $O(\delta^2 + \Upsilon \cdot d_a^2 \cdot \delta)$, while we usually have $\delta^2 \ll \Upsilon \cdot d_a^2 \cdot \delta$ in practice.

However, it is unnecessary to conduct $O(\Upsilon \cdot d_a^2)$ distance queries. In particular, DeAsyn and InAsyn may update a certain label $O(d_a)$ times, each time for a smaller distance value. For example, in Figure 3, for a certain $j \in [d_a + 2, \Upsilon]$, DeAsyn and RepairedDeAsyn sequentially update $L(v_j)[v_0] = M + 1 + d_a - i + 2$ through the path between $v_i$ and $v_j$ for each $i \in [2, d_a + 1]$, as these paths that contain larger numbers of edges have smaller total weights. If we update a certain label just once, then we could accelerate the maintaining process by only conducting $O(\Upsilon \cdot d_a)$ distance queries. We cannot achieve this goal by trivially modifying DeAsyn or InAsyn. The following FastDeM and FastInM overcome this challenge by applying a novel and non-trivial hub-sorted label update process with distance query result memorization, without increasing the space complexity.

## 4 THE PROPOSED FastDeM ALGORITHM

In this section, we propose a fast edge weight decrease maintenance algorithm: FastDeM (Algorithm 3) to update $L$ and *PPR* after an edge weight decrease. Suppose that $w(a, b)$ decreases from $w_0$ to $w_1$, *i.e.*, $w_0 > w_1$. It updates all outdated labels in an original hub-sorted way with distance query result memorization. We will first show the details of FastDeM, and then discuss the differences between FastDeM and related algorithms, in this section.

**The FastDeM algorithm:** The algorithm inputs the updated graph $G(V, E, w)$, $L$, *PPR* and $(a, b)$. First, it uses $CL^c$ to record to-be-updated labels of $a$ and $b$ via a similar process with Algorithm 2 (Line 1). The difference is that it does not update $L$ during this process. Then, starting from recorded labels in $CL^c$, it calls *DIFFUSE* to update labels (Line 2). After that, FastDeM returns $L$ and *PPR* (Line 3). The details of *DIFFUSE* are as follows.

*DIFFUSE* processes each $(u, v, d_u) \in CL^c$ (Line 4), and generates new labels with the same hub $v$ as follows. First, it initializes $Dis[u] = d_u$, $Dis[s] = -1$ for every $s \in V \setminus u$, and a min priority queue $Q$ that contains an element of $u$ with the priority of $d_u$ (Line 5). Note that, in both Algorithm 1 and *DIFFUSE*, priorities in $Q$ are to-be-updated label-contained distance values. $d$ in Line 3 of Algorithm 1 records priorities in $Q$. Differently, here, $Dis$ records queried distances between $v$ and other vertices, which may not equal priorities in $Q$ in the following process.

While $Q \neq \emptyset$ (Line 6), *DIFFUSE* pops the top element $x$ with the priority of $d_x$, and updates a label $L(x)[v] = d_x$ (Line 7). Then, for each $x_n \in N(x)$ such that $r(v) > r(x_n)$ (Line 8), it checks whether $Dis[x_n] == -1$. If $Dis[x_n] == -1$, *i.e.*, $Dis[x_n]$ has not been initialized yet, it sets $Dis[x_n]$ to be the queried distance between $x_n$ and $v$ (Line 9). The initialization of $Dis$ to be queried distances is different from the initialization of $d$ in Algorithm 1.

If $Dis[x_n] > d_x + w(x_n, x)$, it updates $Dis[x_n]$ to be $d_x + w(x_n, x)$, and inserts (or updates) the element of $x_n$ into $Q$ with the priority of $d_x + w(x_n, x)$ (Line 11). Otherwise, it performs this insertion or update when $v \in C(x_n)$ & $\min\{L(x_n)[v], Q(x_n)\} > d_x + w(x_n, x)$ (Line 13), where $Q(x_n)$ is the priority of $x_n$ in $Q$. This step is to update all outdated labels, no matter whether the queried distance is larger than the to-be-updated distance or not. Due to this step, $Dis[x_n]$ may be smaller than $Q(x_n)$, *e.g.*, for the update of $(v_1, 6.6) \in L(v_5)$ after the 5th change in Figure 2, $Dis[v_5] = 6$, while $Q(v_5) = 6.6$. Notably, such updated labels do not induce incorrect query results. The purpose of such updates is to ensure that all the outdated labels are updated to be direct spreads of other labels via updated edges, and as a result can be updated by a later edge weight increase maintenance when required, since InAsyn and FastInM identify outdated labels by mainly checking whether these labels are direct spreads of other outdated labels, as discussed in Section 3.1.

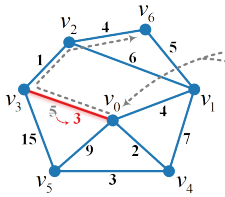
After performing the above step, it inserts $v$ into $PPR[x_n, h_c]$, and also inserts $x_n$ into $PPR[v, h_c]$ (Line 15), where $h_c$ is the common hub responsible for $Dis[x_n]$. Specifically, if $Dis[x_n]$ is updated in

**Algorithm 3** The FastDeM algorithm

**Input:** the updated graph $G(V, E, w)$, $L$, $PPR$, $(a, b)$ // $w_0 > w_1$
**Output:** the maintained $L$ and $PPR$

1: $CL^c = \emptyset$; Conduct Lines 2-17 of Algorithm 2 without updating $L$
2: $DIFFUSE(CL^c)$
3: Return $L$ and $PPR$

**Procedure** $DIFFUSE(CL^c)$
4: **for** each $(u, v, d_u) \in CL^c$ **do**
5: $Dis[u] = d_u$, $Dis[s] = -1\ \forall s \in V \setminus u$, $Q = \{(u \mid d_u)\}$
6: **while** $Q \neq \emptyset$ **do**
7: Pop $(x \mid d_x)$ out of $Q$, $L(x)[v] = d_x$
8: **for** each $x_n \in N(x)$ such that $r(v) > r(x_n)$ **do**
9: **if** $Dis[x_n] == -1$ **then** $Dis[x_n] = Query(x_n, v, L)$
10: **if** $Dis[x_n] > d_x + w(x_n, x)$ **then**
11: $Dis[x_n] = d_x + w(x_n, x)$; Insert (or update) $(x_n \mid Dis[x_n]) \in Q$
12: **else**
13: **if** $v \in C(x_n)$ & $\min\{L(x_n)[v], Q(x_n)\} > d_{new} = d_x + w(x_n, x)$ **then**
14: Insert (or update) $(x_n \mid d_{new}) \in Q$
15: $PPR[x_n, h_c].push(v), PPR[v, h_c].push(x_n)$




| | | | | |
|---|---|---|---|---|
| $L(v_0)$ : | $(v_0, 0)$ | | | |
| $L(v_1)$ : | $(v_0, 4)$ | $(v_1, 0)$ | | |
| $L(v_2)$ : | $(v_0, 4)$ | $(v_1, 6)$ | $(v_2, 0)$ | |
| $L(v_3)$ : | $(v_0, 3)$ | $(v_1, 7)$ | $(v_2, 1)$ | $(v_3, 0)$ |
| $L(v_4)$ : | $(v_0, 2)$ | $(v_4, 0)$ | | |
| $L(v_5)$ : | $(v_0, 5)$ | $(v_4, 3)$ | $(v_5, 0)$ | |
| $L(v_6)$ : | $(v_0, 8)$ | $(v_1, 5)$ | $(v_2, 4)$ | $(v_6, 0)$ |

**Figure 4: An example of** FastDeM.

Line 9, then $h_c$ is the common hub for the queried distance between $x_n$ and $v$, or if $Dis[x_n]$ is updated in Line 11, then $h_c = v$.

**An example of** FastDeM**:** Consider the graph in Figure 1. Suppose that $w(v_0, v_3)$ decreases from 5 to 3. We use Figure 4 to illustrate the maintaining process of FastDeM. Initially, it pushes $(v_3, v_0, 3)$ into $CL^c$. Subsequently, it calls $DIFFUSE$ to sequentially generate $(v_0, 3) \in L(v_3)$, $(v_0, 4) \in L(v_2)$ and $(v_0, 8) \in L(v_6)$. It also pushes $v_0$ into $PPR[v_1, v_0]$, $PPR[v_2, v_0]$, $PPR[v_3, v_0]$, $PPR[v_5, v_0]$, and pushes $v_0, v_1, v_2, v_3, v_5$ into $PPR[v_0, v_0]$ during the above process.

**The correctness of** FastDeM**:** We show that FastDeM can maintain 2-hop labels after an edge weight decrease as follows.

Theorem 1. *Given a graph $G$, a set $L$ of labels that satisfies the 2-hop cover constraint, and the corresponding PPR, suppose that $w(a, b)$ decreases from $w_0$ to $w_1$, then* FastDeM *can maintain $L$ & $PPR$ such that we can use the maintained labels to correctly query the shortest distance between every pair of vertices on the updated graph.*

Proof. Consider a pair of vertices $s$ and $t$, we use $p(s, t)$ and $p'(s, t)$ to denote a shortest path between $s$ and $t$ before and after the edge weight change, respectively. Let $u$ and $u'$ be the vertices with the highest rank in all shortest paths between $s$ and $t$ before and after the change, respectively, and $u \in p(s, t)$ and $u' \in p'(s, t)$. Also let $d(s, u')$ and $d(t, u')$ be the shortest distances between $s$ and $u'$, and between $t$ and $u'$, respectively, before the change. Similarly, let $d'(s, u')$ and $d'(t, u')$ be the distances after the change. We prove that we can use the maintained $L$ to correctly query the shortest distance between $s$ and $t$ on the updated graph as follows.

There are two cases: 1) $(a, b) \in p'(s, t)$; and 2) $(a, b) \notin p'(s, t)$.

Case 1: $(a, b) \in p'(s, t)$. Without loss of generality, suppose that $a$ is closer to $s$ than $b$ along $p'(s, t)$, and $u'$ is between $(a, b)$ and $t$, as illustrated in Figure 5 (a). We have $d(t, u') = d'(t, u')$, and $u'$ is the vertex with highest rank in all shortest paths between $u'$ and $t$ both before and after the change. Thus, hub $u'$ spreads from itself to $t$ along $p'(u', t)$ before the change, as otherwise there must be a

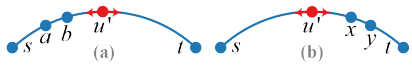



**Figure 5: Some illustrations for the correctness proofs.**

vertex with a higher rank than $u'$ in a shortest path between $u'$ and $t$ that can prune this spread. Therefore, $(u', d'(t, u')) \in L(t)$, and similarly $(u', d'(b, u')) \in L(b)$, both before and after the change. FastDeM calls $DIFFUSE$ to re-spread hub $u'$ from $b$ to $s$ along $p'(s, t)$. Thus, $(u', d'(s, u')) \in L(s)$ after the maintenance, and we can use the maintained $L$ to correctly query $d'(s, t)$.

Case 2: $(a, b) \notin p'(s, t)$. $d(s, t) = d'(s, t)$ and $u = u'$. Assume that $u'$ does not spread from $u'$ to $t$ along $p'(u', t)$, and let $x$ be the vertex farthest to $u'$ along $p'(u', t)$ such that $u'$ spreads from $u'$ to $x$ along $p'(u', t)$, and also let $y$ be the neighbor of $x$ along $p'(u', t)$ such that $u'$ does not spread from $u'$ to $y$ along $p'(u', t)$, as shown in Figure 5 (b). To ensure that $u'$ does not spread to $y$ along $p'(u', t)$, there must be a vertex $z$ such that $z \in C(u') \cap C(y)$, $r(z) > r(u')$, and $z$ is in a shortest path between $u'$ and $y$. This contradicts with the assumption that $u'$ has the highest rank in all shortest paths between $s$ and $t$. Thus, $u'$ spreads to $y$, and ultimately to $t$, along $p'(u', t)$. Similarly, $u'$ spreads to $s$ along $p'(u', s)$. Thus, we can use the maintained $L$ to correctly query $d'(s, t)$. This theorem holds. □

**The complexities of** FastDeM**:** FastDeM has a time complexity of

$$O\Big(\delta^2 + \Upsilon \cdot (\log \Upsilon + d_a \cdot \delta)\Big),$$

where $\delta$ is the average number of labels associated with each vertex. Like DeAsyn, FastDeM has a space complexity of $O(|E| \cdot \delta)$, since $|L| + |PPR| = O(|E| \cdot \delta)$. The details are in the supplement *.

**Differences between** FastDeM **and related algorithms:** There are mainly two related algorithms: DePLL [4] and DeAsyn [27]. Both algorithms update labels in an iterative hub-assorted way, *i.e.,* the processes of updating labels with different hubs mix with each other. DePLL does not, while DeAsyn does, utilize the vertex rank pruning technique, *e.g.,* Line 3 in Algorithm 2. As a result, DePLL is slower than DeAsyn. Differently, FastDeM updates labels via a novel hub-sorted process with distance query result memorization. Specifically, $DIFFUSE$ only generates labels with hub $v$ in each iteration, and applies $Dis$ to record and update some distance query results, without actually conducting these queries, *e.g.,* the update of $Dis[x_n]$ in Line 11 does not require a distance query. There is no such query result memorization in DePLL or DeAsyn. Notice that, $Dis$ does not increase the space complexity, since the size of $Dis$ is negligible when comparing to $L$ and $PPR$. Furthermore, $Dis$ is different from $d$ in Algorithm 1, since $d$ records to-be-updated distance values, *i.e.,* priorities in $Q$, while $Dis$ does not record priorities in $Q$, but records queried distances, which could be smaller than to-be-updated distance values, as discussed in the description of Algorithm 3. Due to this new memorization technique, different from existing 2-hop label maintenance [4, 8, 27] and generation [3, 9, 12] algorithms (including DePLL, DeAsyn and Algorithm 1) that conduct a distance query directly before each label update, FastDeM dissociates distance queries from label updates, and conducts $O(\Upsilon \cdot d_a)$ distance queries, *e.g.,* for a certain $j \in [d_a + 2, \Upsilon]$ in Figure 3, it updates $L(v_j)[v_0]$ only once. In comparison, DeAsyn conducts

$O(\Upsilon \cdot d_a^2)$ distance queries, and has a larger time complexity of

$$O\Big(\delta^2 + \Upsilon \cdot d_a^2 \cdot \delta\Big),$$

since we generally have $\log \Upsilon \ll d_a^2 \cdot \delta$ in practice.

## 5 THE PROPOSED FastInM ALGORITHM

In this section, we propose a fast edge weight increase maintenance algorithm: FastInM (Algorithm 4) to update $L$ and $PPR$ after an edge weight increase. Suppose that the weight of $(a, b)$ increases from $w_0$ to $w_1$, *i.e.,* $w_0 < w_1$. The idea of FastInM is to first deactivate all outdated labels whose inside distance values depend on $w_0$, and then update labels in a similar hub-sorted way with FastDeM.

**The FastInM algorithm:** The algorithm inputs the updated graph $G(V, E, w)$, $L$, $PPR$, $(a, b)$ and $w_0$. First, it initializes three empty sets $AL_1$, $AL_2$ and $AL_3$ (Line 1). It uses $AL_1$ to store labels in $L(a) \cup L(b)$ that correspond to paths that pass through $(a, b)$. Based on $AL_1$, it finds all labels that correspond to paths that pass through $(a, b)$, and then deactivates these labels, and also uses $AL_2$ to record these labels. After the deactivation, some originally pruned labels can be newly produced. It uses $AL_3$ to record these labels, and uses these labels as starting points to generate more labels.

After initializing $AL_1$, $AL_2$ and $AL_3$, the algorithm populates $AL_1$ as follows. For each $(v, d_{va}) \in L(a)$, it checks whether $r(v) \geq r(b)$ & $(v, d_{va} + w_0) \in L(b)$ (Line 3). If $r(v) \geq r(b)$, then $v$ could be a hub in $L(b)$. Furthermore, $(v, d_{va} + w_0) \in L(b)$ corresponds to a path that passes through $(a, b)$. The algorithm pushes $(b, v, d_{va} + w_0)$ into $AL_1$ to record this label. Symmetrically, it processes each $(v, d_{vb}) \in L(b)$ via a similar process (Lines 4-5). After that, it uses $SPREAD_1$ to populate $AL_2$ based on $AL_1$, uses $SPREAD_2$ to populate $AL_3$ based on $AL_2$, and then uses $SPREAD_3$ to generate new labels based on $AL_3$ (Line 6). Ultimately, it returns the maintained $L$ and $PPR$ (Line 7). We describe the above 3 procedures as follows.

$SPREAD_1$ processes each $(u, v, d') \in AL_1$ (Line 8) as follows. It initializes a queue with an element $(u, d')$. While this queue is not empty, it pops $(x, d_x)$ out of the queue, sets $L(x)[v] = \infty$, and then pushes $(x, v)$ into $AL_2$ (Line 11). Notably, the original label $(v, d_x) \in L(x)$ corresponds to a path that passes through $(a, b)$. The above step is to deactivate this label by setting the inside distance value to $\infty$. Subsequently, for each $x_n \in N(x)$ such that $r(v) > r(x_n)$ (Line 12), if $(v, d_x + w(x, x_n)) \in L(x_n)$, which indicates that $(v, d_x + w(x, x_n)) \in L(x_n)$ also corresponds to a path that passes through $(a, b)$, then it pushes this label into the queue (Line 13). After the while loop, $SPREAD_1$ deactivates all labels that correspond to paths that pass through $(a, b)$, and uses $AL_2$ to record these labels.

$SPREAD_2$ processes each $(x, y) \in AL_2$ (Line 14) via the following steps. As $L(x)[y]$ has been set to $\infty$, some labels that are originally pruned by $L(x)[y]$ may be newly generated. To identify such labels, it enumerates each $t \in PPR[x, y] \cup y$ (Line 15). If $t \in PPR[x, y]$, then either $y \in C(x) \cap C(t)$ is responsible for originally pruning the spread of hub $t$ from a neighbor of $x$ to $x$ (*i.e.,* $r(t) > r(x)$), or $y$ is responsible for originally pruning the spread of hub $x$ from a neighbor of $t$ to $t$ (*i.e.,* $r(x) > r(t)$). If $t = y$ (*i.e.,* $r(t) > r(x)$), then it may be possible to update $L(x)[y]$ by re-spreading hub $y$ from a neighbor of $x$ to $x$. Based on these analyses, it identifies to-be-generated labels as follows. If $r(t) > r(x)$, it computes the minimum distance for spreading hub $t$ from a neighbor of $x$ to $x$:

$$d1(x, t) = \min_{x_n \in N(x)} \{L(x_n)[t] + w(x, x_n)\}. \quad (2)$$

If the queried distance between $x$ and $t$ is larger than $d1(x, t)$, then it pushes $(x, t, d1(x, t))$ into $AL_3$ (Line 18), which means that $L(x)[t] = d1(x, t)$ could be newly generated. Otherwise, it inserts $t$ into $PPR[x, h_c]$, and inserts $x$ into $PPR[t, h_c]$ (Line 19), where $h_c$ is the common hub for the above queried distance. If $r(x) > r(t)$, it populates $AL_3$ and $PPR$ similarly (Lines 20-23).

$SPREAD_3$ processes each $(u, v, d_u) \in AL_3$ (Line 24), and generates new labels with the same hub $v$ starting from $L(u)[v]$ in a similar way with $DIFFUSE$ in FastDeM. A difference is that $SPREAD_3$ first checks whether the queried distance between $u$ and $v$ is no larger than $d_u$ (Line 25), since newly generated labels could prune the generation of other labels. Another difference is that, when $Dis[x_n] \leq d_x + w(x_n, x)$, $SPREAD_3$ does not insert (or update) the element of $x_n$ into $Q$ with the priority of $d_x + w(x_n, x)$, since all the outdated labels have been deactivated in $SPREAD_1$.

**Algorithm 4** The FastInM algorithm

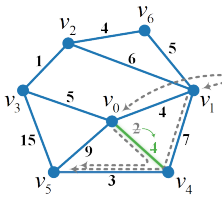
**Input:** the updated graph $G(V, E, w)$, $L$, $PPR$, $(a, b)$, $w_0$ // $w_0 < w_1$
**Output:** the maintained $L$ and $PPR$

1: $AL_1 = AL_2 = AL_3 = \emptyset$
2: **for** each label $(v, d_{va}) \in L(a)$ **do**
3: **if** $r(v) \geq r(b)$ & $(v, d_{va} + w_0) \in L(b)$ **then** $AL_1.push((b, v, d_{va} + w_0))$
4: **for** each label $(v, d_{vb}) \in L(b)$ **do**
5: **if** $r(v) \geq r(a)$ & $(v, d_{vb} + w_0) \in L(a)$ **then** $AL_1.push((a, v, d_{vb} + w_0))$
6: $SPREAD_1(AL_1, AL_2), SPREAD_2(AL_2, AL_3), SPREAD_3(AL_3)$
7: Return $L$ and $PPR$

**Procedure** $SPREAD_1(AL_1, AL_2)$
8: **for** each $(u, v, d') \in AL_1$ **do**
9: $Queue = \{(u, d')\}$
10: **while** $Queue \neq \emptyset$ **do**
11: $Queue.pop((x, d_x)), L(x)[v] = \infty, AL_2.push((x, v))$
12: **for** each $x_n \in N(x)$ such that $r(v) > r(x_n)$ **do**
13: **if** $(v, d_x + w(x, x_n)) \in L(x_n)$ **then** $Queue.push((x_n, d_x + w(x, x_n)))$

**Procedure** $SPREAD_2(AL_2, AL_3)$
14: **for** each $(x, y) \in AL_2$ **do**
15: **for** each $t \in PPR[x, y] \cup y$ **do**
16: **if** $r(t) > r(x)$ **then**
17: $d1(x, t) = \min_{x_n \in N(x)} \{L(x_n)[t] + w(x, x_n)\}$
18: **if** $Query(x, t, L) > d1(x, t)$ **then** $AL_3.push((x, t, d1(x, t)))$
19: **else** $PPR[x, h_c].push(t), PPR[t, h_c].push(x)$
20: **if** $r(x) > r(t)$ **then**
21: $d1(t, x) = \min_{t_n \in N(t)} \{L(t_n)[x] + w(t, t_n)\}$
22: **if** $Query(t, x, L) > d1(t, x)$ **then** $AL_3.push((t, x, d1(t, x)))$
23: **else** $PPR[t, h_c].push(x), PPR[x, h_c].push(t)$

**Procedure** $SPREAD_3(AL_3)$
24: **for** each $(u, v, d_u) \in AL_3$ **do**
25: **if** $Q(u, v, L) \leq d_u$ **then** $PPR[u, h_c].push(v), PPR[v, h_c].push(u)$; Continue
26: $Dis[u] = d_u, Dis[s] = -1$ for each $s \in V \setminus u, Q = \{(u \mid d_u)\}$
27: **while** $Q \neq \emptyset$ **do**
28: Pop $(x \mid d_x)$ out of $Q$, $L(x)[v] = \min(d_x, L(x)[v])$
29: **for** each $x_n \in N(x)$ such that $r(v) > r(x_n)$ **do**
30: **if** $Dis[x_n] == -1$ **then** $Dis[x_n] = Query(x_n, v, L)$
31: **if** $Dis[x_n] > d_x + w(x_n, x)$ **then**
32: $Dis[x_n] = d_x + w(x_n, x)$; Insert (or update) $(x_n \mid Dis[x_n]) \in Q$
33: **else** $PPR[x_n, h_c].push(v), PPR[v, h_c].push(x_n)$




| | | | | | |
|---|---|---|---|---|---|
| $L(v_0)$ : | $(v_0, 0)$ | | | | |
| $L(v_1)$ : | $(v_0, 4)$ | $(v_1, 0)$ | | | |
| $L(v_2)$ : | $(v_0, 6)$ | $(v_1, 6)$ | $(v_2, 0)$ | | |
| $L(v_3)$ : | $(v_0, 5)$ | $(v_1, 7)$ | $(v_2, 1)$ | $(v_3, 0)$ | |
| $L(v_4)$ : | $(v_0, 4)$ | $(v_1, 7)$ | $(v_4, 0)$ | | |
| $L(v_5)$ : | $(v_0, 7)$ | $(v_1, 10)$ | $(v_4, 3)$ | $(v_5, 0)$ | |
| $L(v_6)$ : | $(v_0, 9)$ | $(v_1, 5)$ | $(v_2, 4)$ | $(v_6, 0)$ | |

**Figure 6: An example of FastInM.**

**An example of** FastInM**:** For the graph in Figure 1, suppose that $w(v_0, v_4)$ increases from 2 to 4. We use Figure 6 to illustrate the process of FastInM. First, it pushes $(v_4, v_0, 2)$ into $AL_1$. Subsequently, it conducts $SPREAD_1$ to (i) deactivate $(v_0, 2) \in L(v_4)$ and $(v_0, 5) \in L(v_5)$; and (ii) push $(v_4, v_0)$ and $(v_5, v_0)$ into $AL_2$. Then, it calls $SPREAD_2$ to generate $AL_3$ based on $AL_2$. For $(v_4, v_0) \in AL_2$, since $PPR[v_4, v_0] = \{v_1\}$, it pushes $(v_4, v_0, 4)$ and $(v_4, v_1, 7)$ into $AL_3$. For $(v_5, v_0) \in AL_2$, since $PPR[v_5, v_0] = \{v_1, v_2, v_3\}$, it pushes $(v_5, v_0, 9)$, $(v_5, v_1, 22)$, $(v_5, v_2, 16)$ and $(v_5, v_3, 15)$ into $AL_3$. After that, it calls $SPREAD_3$ to generate new labels from the starting points in $AL_3$ as follows. First, for $(v_4, v_0, 4) \in AL_3$, it generates two new labels $(v_0, 4) \in L(v_4)$ and $(v_0, 7) \in L(v_5)$. Similarly, for $(v_4, v_1, 7) \in AL_3$, it generates two new labels $(v_1, 7) \in L(v_4)$ and $(v_1, 10) \in L(v_5)$. On the other hand, for other elements in $AL_3$, it does not generate new labels. Like Algorithm 3, it also pushes new elements into $PPR$ in the above processes of generating new labels.

**The correctness of** FastInM**:** We show that FastInM can maintain 2-hop labels for an edge weight increase via the following theorem.

Theorem 2. *Given a graph $G$, a set $L$ of labels that satisfies the 2-hop cover constraint, and the corresponding PPR, suppose that $w(a, b)$ increases from $w_0$ to $w_1$, then* FastInM *can maintain $L$ and PPR such that we can use the maintained labels to correctly query the shortest distance between every pair of vertices on the updated graph.*

Proof. Like the proof of Theorem 1, we consider two cases: 1) $(a, b) \in p'(s, t)$; and 2) $(a, b) \notin p'(s, t)$.

Case 1: $(a, b) \in p'(s, t)$. $p(s, t) = p'(s, t)$ and $u = u'$. Without loss of generality, suppose that $a$ is closer to $s$ than $b$ along $p(s, t)$, and $u$ is between $(a, b)$ and $t$, as shown in Figure 5 (a). Before the change, if $u \notin C(s)$, then there must be a vertex $x \in C(s) \cap C(u)$ such that $r(x) > r(u)$ and $x$ is in a shortest path between $s$ and $t$. This contradicts with the assumption that $u$ has the highest rank in all shortest paths between $s$ and $t$. Thus, $u \in C(s)$, and similarly $u \in C(t)$. Specifically, $(u, d(s, u)) \in L(s)$ and $(u, d(t, u)) \in L(t)$ before the maintenance. Since $d(s, u) < d'(s, u)$, FastInM first calls $SPREAD_1$ and $SPREAD_2$ to deactivate $(u, d(s, u)) \in L(s)$, and then calls $SPREAD_3$ to update it to $(u, d'(s, u)) \in L(s)$. On the other hand, since $d(t, u) = d'(t, u)$, FastInM keeps $(u, d(t, u)) \in L(t)$. Thus, we can use the maintained $L$ to correctly query $d'(s, t)$.

Case 2: $(a, b) \notin p'(s, t)$. Before the change, assume that hub $u'$ does not spread from $u'$ to $t$ along $p'(u', t)$, and let $x$ be the vertex farthest to $u'$ along $p'(u', t)$ such that hub $u'$ spreads from $u'$ to $x$ along $p'(u', t)$, and also let $y$ be the neighbor of $x$ along $p'(u', t)$ such that $u'$ does not spread from $u'$ to $y$ along $p'(u', t)$, as shown in Figure 5 (b). To ensure that $u'$ does not spread from $u'$ to $y$ along $p'(u', t)$, there is a vertex $z$ such that $z \in C(u') \cap C(y)$, $r(z) > r(u')$, $z$ is in a shortest path between $u'$ and $y$ before the change, and $u' \in PPR[y, z]$ and $y \in PPR[u', z]$. Since $u'$ is the vertex with the highest rank in all shortest paths between $u'$ and $y$ after the change, $z$ cannot prune the spread of hub $u'$ to $y$ along $p'(u', t)$ any more. Since the length of $p'(u', y)$ does not change due to the graph change, either $L(u')[z]$ or $L(y)[z]$ increases after the graph change, otherwise $z \in C(u') \cap C(y)$ still prunes the spread of hub $u'$ to $y$ along $p'(u', t)$. Thus, FastInM must call $SPREAD_1$ to set either $L(u')[z]$ or $L(y)[z]$ to $\infty$. In either case, using the above $PPR$ information, FastInM performs $SPREAD_2$ and $SPREAD_3$ to re-spread hub $u'$ from $x$ to $y$, and ultimately to $t$, along $p'(s, t)$. Similarly, after the maintenance, hub $u'$ also spreads from $u'$ to $s$ along $p'(s, t)$. Thus, we can use the maintained $L$ to correctly query $d'(s, t)$. Therefore, this theorem holds. □

**The complexities of** FastInM**:** FastInM has a time complexity of

$$O\Big(\Upsilon \cdot \Big(\log \Upsilon + d_a \cdot \delta + \kappa \cdot (d_a + \delta)\Big)\Big),$$

where $\kappa$ is the average number of $PPR$ elements of each vertex-hub pair. Like InAsyn and FastDeM, FastInM has a space complexity of $O(|E| \cdot \delta)$. The details are in the supplement *.

**Differences between** FastInM **and related algorithms:** There are two related algorithms: InPLL [8] and InAsyn [27]. InPLL maintains 2-hop labels for an edge weight increase by (i) heuristically identifying a super set of vertices that may be hubs in outdated labels; (ii) removing possibly outdated labels with hubs inside this super set; and (iii) generating new labels by conducting a breadth-first-search over the whole graph starting from each vertex in this super set. Given that InPLL may heuristically put $O(|V|)$ vertices into this super set; and each breadth-first-search takes $O(|E|)$ time, InPLL has a large time complexity of $O(|V| \cdot |E|)$. The inefficiency of InPLL owes to the fact that it cannot efficiently identify labels affected by edge weight increases, due to the curse of pruning power [27]. In comparison, both InAsyn and FastInM address this issue by employing $PPR$ to efficiently identify affected labels. On the other hand, like the comparison between DeAsyn and FastDeM, FastInM is different from InAsyn in that FastInM updates labels via a novel hub-sorted process with distance query result memorization. As a result, FastInM conducts $O(\Upsilon \cdot d_a)$ distance queries, while InAsyn conducts $O(\Upsilon \cdot d_a^2)$ queries, and has a larger time complexity of

$$O\Big(\Upsilon \cdot \Big(d_a^2 \cdot \delta + \kappa \cdot (d_a + \delta)\Big)\Big),$$

as we generally have $\log \Upsilon \ll d_a^2 \cdot \delta$ in practice.

## REFERENCES


[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*. Springer, 230–241.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*. Springer, 24–35.

[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceedings of the 23rd international conference on World wide web*. 237–248.

[5] Tanya Y Berger-Wolf and Jared Saia. 2006. A framework for analysis of dynamic social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 523–528.

[6] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: Efficient distance querying on road networks by projected vertex separators. In *Proceedings of the 2021 International Conference on Management of Data*. 313–325.

[7] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and distance queries via 2-hop labels. In *Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete Algorithms*. 937–946.

[8] Gianlorenzo D'angelo, Mattia D'emidio, and Daniele Frigioni. 2019. Fully dynamic 2-hop cover labeling. *Journal of Experimental Algorithmics* 24 (2019), 1–36.

[9] Ruoming Jin, Zhen Peng, Wendell Wu, Feodor Dragan, Gagan Agrawal, and Bin Ren. 2020. Parallelizing pruned landmark labeling: dealing with dependencies in graph algorithms. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–13.

[10] Theodoros Lappas, Kun Liu, and Evimaria Terzi. 2009. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 467–476.

[11] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2016. Efficient and progressive group Steiner tree search. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 91–106.

[12] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling distance labeling on small-world networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1060–1077.

[13] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2020. Scaling up distance labeling on graphs with core-periphery properties. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1367–1381.

[14] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2021. Distance labeling: on parallelism, compression, and ordering. *The VLDB Journal* 31, 1 (2021), 129–155.

[15] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457.

[16] Anirban Majumder, Samik Datta, and KVM Naidu. 2012. Capacitated team formation problem on social networks. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1005–1013.

[17] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data*. 709–724.

[18] Dian Ouyang, Dong Wen, Lu Qin, Lijun Chang, Xuemin Lin, and Ying Zhang. 2023. When hierarchy meets 2-hop-labeling: efficient shortest distance and path queries on road networks. *The VLDB Journal* (2023), 1–25.

[19] Vedran Sekara, Arkadiusz Stopczynski, and Sune Lehmann. 2016. Fundamental structures of dynamic social networks. *Proceedings of the national academy of sciences* 113, 36 (2016), 9977–9982.

[20] Yuxuan Shi, Gong Cheng, and Evgeny Kharlamov. 2020. Keyword search over knowledge graphs via static and dynamic hub labelings. In *Proceedings of The Web Conference*. 235–245.

[21] Han Hee Song, Tae Won Cho, Vacha Dave, Yin Zhang, and Lili Qiu. 2009. Scalable proximity estimation and link prediction in online social networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. 322–335.

[22] Yahui Sun, Xiaokui Xiao, Bin Cui, Saman Halgamuge, Theodoros Lappas, and Jun Luo. 2021. Finding Group Steiner Trees in Graphs with both Vertex and Edge Weights. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1137–1149.

[23] Xinyu Wang, Zhou Zhao, and Wilfred Ng. 2016. Ustf: A unified system of team formation. *IEEE Transactions on Big Data* 2, 1 (2016), 70–84.

[24] Shuang Yang, Yahui Sun, Jiesong Liu, Xiaokui Xiao, Rong-Hua Li, and Zhewei Wei. 2022. Approximating probabilistic group Steiner trees in graphs. *Proceedings of the VLDB Endowment* 16, 2 (2022), 343–355.

[25] Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. 2022. Shortest-path queries on complex networks: experiments, analyses, and improvement. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2640–2652.

[26] Junhua Zhang, Long Yuan, Wentao Li, Lu Qin, and Ying Zhang. 2021. Efficient label-constrained shortest path queries on road networks: a tree decomposition approach. *Proceedings of the VLDB Endowment* (2021).

[27] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-hop labeling maintenance in dynamic small-world networks. In *2021 IEEE 37th International Conference on Data Engineering*. IEEE, 133–144.

[28] Mengxuan Zhang, Lei Li, Goce Trajcevski, Andreas Zufle, and Xiaofang Zhou. 2023. Parallel Hub Labeling Maintenance With High Efficiency in Dynamic Small-World Networks. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[29] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An experimental evaluation and guideline for path finding in weighted dynamic network. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2127–2140.